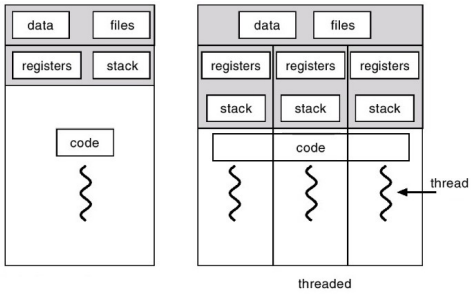
# Graduate Operating Systems
## (Threads & Events)

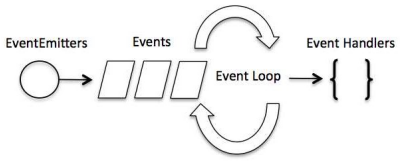
Fall 2020

## Today's Papers

- **[6]** Rob von Behren, Jeremy Condit, and Eric Brewer, "Why Events are a Bad Idea (for high-concurrency servers)", Workshop on Hot Topics in Operating Systems, 2003.
- **[7]** Matt Welsh, David Culler, and Eric Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services", ACM Symposium on Operating Systems Principles, 2001.

# Threads vs. Events



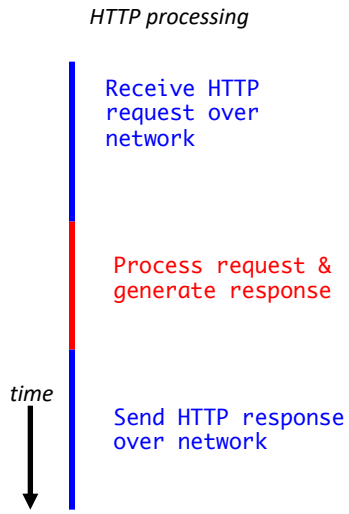




# Threads vs. Events

- 1995: *Why Threads are a Bad Idea (for most purposes)*
  - John Ousterhout (UC Berkeley, Sun Labs)
- 2001: *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services*
  - Staged, Event-driven Architecture
  - M. Welsh, D. Culler, and Eric Brewer (UC Berkeley)
- 2003: *Why Events are a Bad Idea (for high-concurrency servers)*
  - R. van Behren, J. Condit, Eric Brewer (UC Berkeley)

# Background

- *How can we scale up servers to handle many simultaneous requests?*

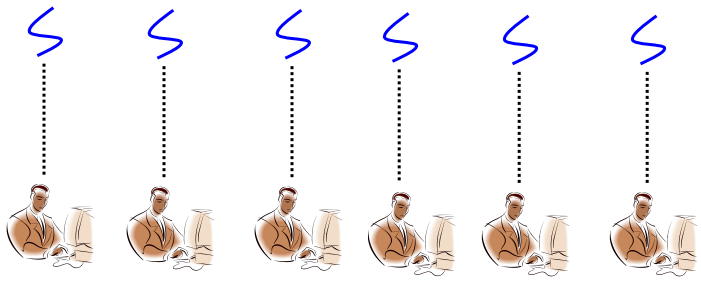



# Quick Example

- *Suppose it takes 1 second for a web server to transfer a file to the client. Of this time, 10 milliseconds is dedicated to CPU processing. How many simultaneous requests do we need to keep the CPU fully utilized?*

## Strategy #1: Thread-per-Request

- Run each web request in its own thread
  - Assuming kernel threads, blocking I/O operations only stall one thread



## Strategy #1: Thread-per-Request

```
while (true) {
    read request from socket
    read requested file into buffer
    write buffer content over socket
    close socket
}
```

## Strategy #2: Event-Driven Execution

- Use a single thread for all requests
- Use non-blocking I/O
  - Replace blocking I/O with calls that return immediately
  - Program is notified about interesting I/O events
- This is philosophically similar to hardware interrupts
  - "Tell me when something interesting happens"

## Strategy #2: Event-Driven Execution

```
while (true) {
    find sockets with active I/O
    Socket sock = getActiveSocket();
    if (sock.isReadable())
        handleReadEvent(sock);
    if (sock.isWriteable())
        handleWriteEvent(sock);
}
```

- Example: GUI frameworks (*What are examples of events?*)

## UNIX “select” System Call

```
int select(int nfds, fd_set *readfds, fd_set
*writefds, fd_set *exceptfds, struct timeval
*timeout);
```

## Events

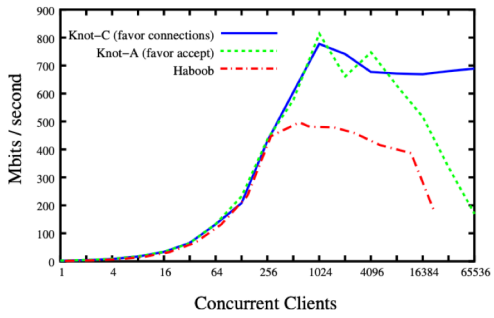
- Are events simpler or harder than threads?
- Events & concurrency
- Is event-based or threaded code easier to understand?

# "Problems" with Threads

- Threads for high concurrency do not perform well
- Threads have restrictive control flow
- Thread synchronization is heavyweight
- Thread stacks are ineffective
- Optimal scheduling decisions are hard

# Compiler Support for Threads

- Dynamic stack growth
- Live state management
- Synchronization

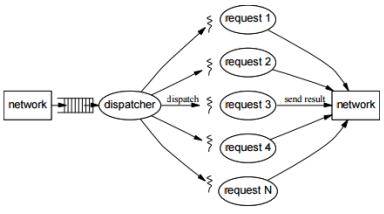


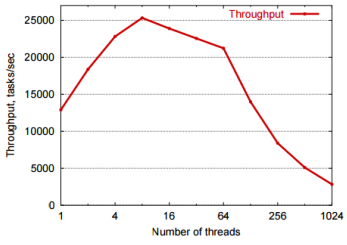
# Paper “SEDA”

- “Slashdot effect”; peak load
- “Well-conditioned service”
  - Throughput: saturate with load
  - Response time: increase linearly with load
  - Graceful degradation

# Thread-Based Concurrency

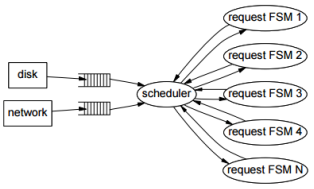
- Easy to program; high concurrency
- Overheads
- Throughput degradation (bounded thread pools)
- Latency

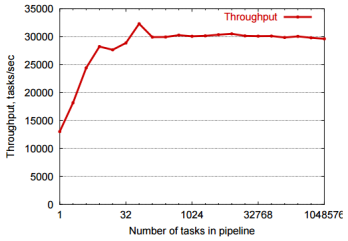


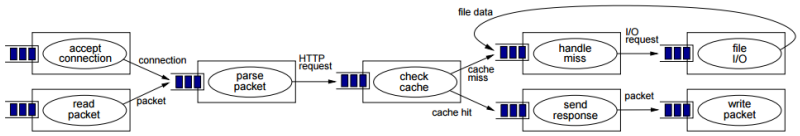
# Event-Driven Concurrency

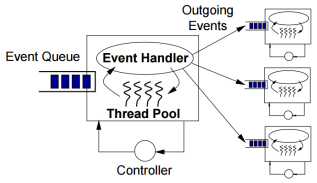
- Small number of threads (typically one per CPU); non-blocking I/O
- Robust to load
- Latency
- Scheduling decisions; load dropping



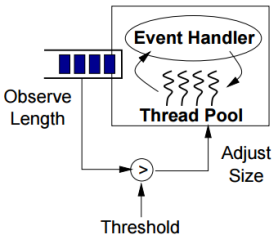

# SEDA: Staged Event Driven Architecture

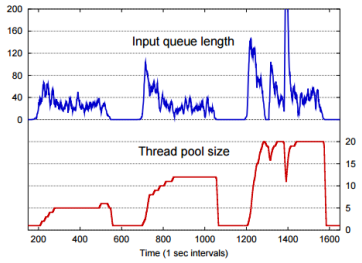


# Resource Controllers

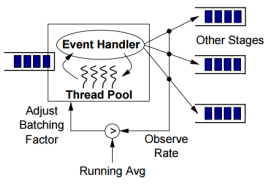
- Thread pool controller
  - Adjust number of threads executing

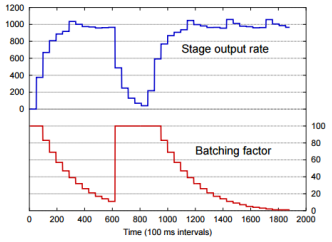



# Resource Controllers

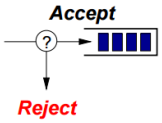
- Batching controller
  - Adjust number of events processed by each iteration of the event handler



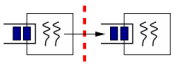



# Queues

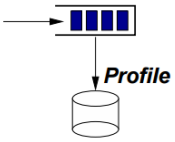
- Queues are finite
  - Enqueuing may fail
  - Block on full queue -> backpressure
  - Drop rejected events -> load shedding



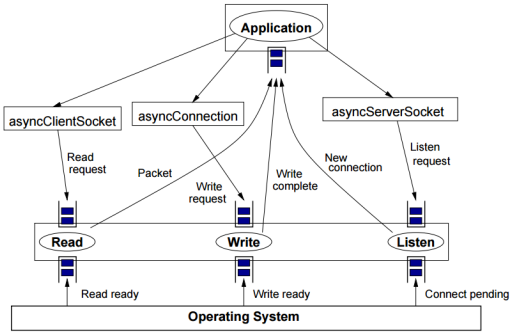

- Queues introduce explicit execution boundaries
  - Threads may only execute within a single stage
  - Performance isolation, modularity, independent load management



- Explicit event delivery support inspection
  - Trace flow of events through application
  - Monitor queue lengths to detect bottleneck




# Asynchronous I/O



# Summary & Discussion

- SEDA: Staged, Event-Driven Architecture
  - Applications consist of **connected stages each serviced by one or more threads**
  - **Dynamic resource controllers** examine and react to high load conditions and control thread usage
- Measurement and control vs. reservation
  - Mechanisms for detecting overload
  - Policies to deal with overload
- SEDA ease of programming
  - Reduced need for synchronization & race conditions
  - Separate stages for different components of application/server